Inference in receiver operating characteristic surface analysis via a trinormal model-based testing approach

Noll, Samuel ; Furrer, Reinhard ; Reiser, Benjamin ; Nakas, Christos T

Abstract: Receiver operating characteristic (ROC) analysis is the methodological framework of choice for the assessment of diagnostic markers and classification procedures in general, in both two-class and multiple-class classification problems. We focus on the three-class problem for which inference usually involves formal hypothesis testing using a proxy metric such as the volume under the ROC surface (VUS). In this article, we develop an existing approach from the two-class ROC framework. We define a hypothesis-testing procedure that directly compares two ROC surfaces under the assumption of the trinormal model. In the case of the assessment of a single marker, the corresponding ROC surface is compared with the chance plane, that is, to an uninformative marker. A simulation study investigating the proposed tests with existing ones on the basis of the VUS metric follows. Finally, the proposed methodology is applied to a dataset of a panel of pancreatic cancer diagnostic markers. The described testing procedures along with related graphical tools are supported in the corresponding R-package trinROC, which we have developed for this purpose.

DOI: https://doi.org/10.1002/sta4.249

PolKA: Polynomial Key-based Architecture for Source Routing in Network Fabrics

Cristina Dominicini and Diego Mafioletti
Federal Institute of Education Science and Technology of Espírito Santo
Espírito Santo, Brazil
cristina.dominicini@ifes.edu.br,
diego.rossi@ifes.edu.br

Ana C. Locateli, Rodolfo Villaca, Magnos Martinello, and Moisés Ribeiro
Federal University of Espírito Santo
Espírito Santo, Brazil
{ana.locateli,rodolfo.villaca}@ufes.br,
magnos@inf.ufes.br, moises@ele.ufes.br

Alexander Gorodnik
School of Mathematics
University of Bristol
Bristol, UK
a.gorodnick@bristol.ac.uk

***Abstract*—Source routing (SR) is a prominent alternative to table-based routing for reducing the number of network states. However, traditional SR approaches, based on Port Switching, still maintain a state in the packet by using a header rewrite operation. The residue number system (RNS) is a promising way of executing fully stateless SR, in which forwarding decisions at core nodes rely on a simple modulo operation over a route label. Nevertheless, such operation over integer arithmetic is not natively supported by commodity network hardware. Thus, we propose a novel RNS-based SR scheme, named PolKA, that explores binary polynomial arithmetic using Galois field (GF) of order 2. We evaluate PolKA in comparison to Port Switching by implementing emulated and hardware prototypes using P4 architecture. Results show that PolKA can achieve equivalent performance, while providing advanced routing features, such as fast failure reaction and agile path migration.**

***Index Terms*—source routing, network fabrics, residue number system, software defined networks, Chinese remainder theorem.**

I. Introduction

In recent years, we have seen a widespread interest in software defined networking (SDN) as a driver of network architectures evolution. In this sense, an emerging architecture for software-defined data centers (DCs) and WANs is the *network fabric* [1], where complex functions are pushed to the edge, while the core provides a simple and efficient data delivery abstraction. One of the main problems is how to select routing paths in the core fabric and load-balance between them in order to adapt to highly variable traffic patterns.

This is far from a trivial challenge due to the scale, dynamics, and high performance requirements of modern networks. A common approach to this challenge is to encode multiple paths in core nodes in the form of forwarding table entries, and then allow the edge to select among the existent paths. However, the resulting designs require large numbers of table entries [2], are constrained by the limited capacity of switch forwarding tables [3], and still restrict path selection [4].

The decoupling of forwarding from other network functions opens the opportunity to rethink the fabric design towards an stateless core network. In this direction, *Source routing (SR)* schemes, where an edge node[1] adds a route label in the packet header to specify an end-to-end path, have been attracting a lot of attention [4], [5]. These schemes allow traffic engineering to dynamically exploit all existing paths to achieve maximum throughput [4]. Also, SR reduces the control signaling and latency related to path setup convergence, as migrating paths is only a matter of changing the state at the source [6]. Moreover, it greatly simplifies the design of core switches [5].

The most traditional way of executing SR is Port Switching, in which the route label represents a ordered list (or stack) of output ports and the forwarding operation is a pop of the first element [5]. Although this approach drastically reduces the burden of managing network states by eliminating tables in core nodes, it still needs to maintain a state in the packet by using a *route label rewrite operation* in every node to update the position in the list. This operation may be costly for packet networks and difficult to implement in optical networks [7].

In addition, a classical problem in SR is how fast it reacts to node or link failures [8]. When a failure is detected, the source calculates a new route label, but this change takes a long time leading to a loss of packets in transit. A solution is to embed alternate paths in the nodes, but it increases the number of network states. An alternative is to embed paths in the packets, but the flat list encoding used by Port Switching does not offer an implicit way of representing additional paths.

In this paper, we want to push to an extreme design choice, and answer the following questions: (i) is it possible to define a fully stateless SR approach (i.e., no state in the core nodes, nor in the packet) in a network fabric that allows agile path selection and fast failure reaction?; and (ii) how to implement such approach in commodity network hardware with equivalent performance to traditional Port Switching?

To this end, we propose a novel SR scheme, named **PolKA (Polynomial Key-based Architecture)**, which explores special properties from the *Residue Number System (RNS) with polynomial arithmetic* using Galois field (GF) of order 2 [9], known as GF(2). These properties guarantee that the node sequence is irrelevant to derive the route label, which remains unchanged throughout all the path [7], [10]. In this scheme, at any core node, the output port is given by the remainder of the binary polynomial division (i.e., a $\mod$ operation) of the route identifier of the packet by the node identifier.

[1]The edge node may be a virtual switch in a server, a hypervisor, a top-of-rack (ToR) switch, or a ingress domain gateway.

978-1-7281-5684-2/20/$31.00 ©2020 IEEE

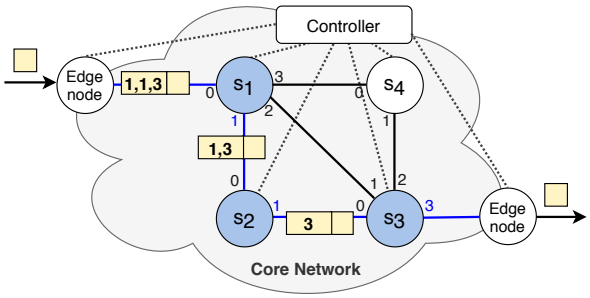


Figure 1: Example of Port Switching SR for a SDN fabric.

In summary, the contributions of this work are: (i) we propose PolKA, a fully stateless RNS-based SR scheme for network fabrics that is compatible with binary polynomial arithmetic; (ii) we propose a technique to enable the implementation of the polynomial $\mod$ operation in P4-enabled programmable switches [11] by reusing Cyclic Redundancy Check (CRC) hardware; (iii) we implement emulated and hardware prototypes to demonstrate that PolKA can achieve similar performance to Port Switching; and (iv) we demonstrate how PolKA can use intrinsic properties from the RNS to enable advanced routing features, like agile path migration, fast failure reaction, and service function chaining (SFC).

This paper is structured as follows. Section II discusses related works. Section III presents the mathematical background of PolKA. Section IV presents PolKA, and performs a scalability analysis. Section V studies how to implement PolKA and Port Switching according to P4 architecture, followed by the implementation and evaluation of proof-of-concept prototypes in Sections VI and VII. Finally, Section VIII discusses conclusions and future works.

II. Related Work

Many works investigated the benefits of SR over traditional table-based routing, such as massive reduction of network states, and optimal use of network capacity [5] [4] [6]. However, most of them use Port Switching, and, therefore, present the previously discussed limitations regarding the need to update the forwarding state in the packet and the difficult to embed alternate paths. Fig. 1 illustrates Port Switching SR for a SDN fabric [5]. When a packet arrives, the edge node communicates with a controller, which calculates a route label that represents an ordered list of ports. In this figure, the list $\{1, 1, 0\}$ maps a path specified by nodes $\{s_1, s_2, s_3\}$ and their respective output ports. Then, each core node pops the first element of this list, modifying the route label of the packet.

In contrast to Port Switching, in an integer RNS-based SR scheme, the controller calculates the route label (*routeID*) as an integer number, which remains the same along the path [10]. It is also responsible for assigning node identifiers to core nodes (*nodeID*s), which must be pairwise co-prime numbers. For example, if we apply this scheme to the scenario of Fig. 1 and assign *nodeID*s 4, 3, 5, 7 to nodes s_1, s_2, s_3, and s_4, respectively, the *routeID* calculated according to RNS is 25. In this way, the output ports for s_1, s_2, and s_3 are, respectively: $(25 \mod 4) = 1$, $(25 \mod 3) = 1$, and $(25 \mod 5) = 0$.

Table I: Comparison of routing methods

Routing method	State at core fabric	Forwarding operation	Commodity network hardware?
Table-based Routing	table	lookup	✓
Port Switching SR	packet	pop	✓
Integer RNS-based SR	-	integer mod	×
PolKA RNS-based SR	-	polynomial mod	✓

Related works on RNS-based SR integrated this scheme with SDN [10], developed fast failure reaction mechanisms [8], investigated techniques to improve the scalability of the *routeID* [12], evaluated latency constraints for multicast in data centers (DCs) [13] [14], and applied the scheme to enable SFC [15]. However, all these works rely on integer RNS arithmetic, and the integer $\mod$ operation cannot be implemented in current commodity network hardware. Therefore, they either use software switches implementations [10], [15], or depend on synthesizing integer division to ASICs or NetFPGAs [14].

PolKA differs from previous RNS-based SR works by bringing a broader SR expressiveness that is closer to elementary binary polynomial operations, introducing a new interpretation over the classical RNS integer arithmetic [7], [10], as Section IV will explain in details. The immediate benefit is to enable the reuse of commodity embedded network functions that are based on polynomial arithmetic. For instance, CRC hardware ordinarily provides wire-speed implementations of a $\mod$ operation with a fixed polynomial base [16], and is evolving for supporting configurable polynomials [17], [18]. Furthermore, we envision that the expressiveness of our polynomial scheme can be extended using GFs of higher orders to enable innovative solutions for modern routing challenges, like network on a chip, network slicing, and multilayer networks.

Other works explored polynomial RNS [9] [19], but haven't applied these concepts to routing. To the best of our knowledge, PolKA is the first work to combine RNS with GF(2) polynomials to solve SR problems, and to propose a technique that allows the reuse of CRC hardware for routing purposes.

Table I summarizes a comparison between routing methods, considering where the *forwarding state* is stored in the core network, what is the *forwarding operation*, and whether this operation can be implemented in *commodity network hardware*. Note that RNS-based SR methods are the only to offer fully stateless routing, because they do not update any state in the route label or in core nodes.

III. Mathematical Background

This section describes the mathematical formulation that supports the proposal of PolKA. More information about finite fields and polynomials rings can be found in [9], [20].

Polynomial Ring over GF(2): Let GF(2) = $\{0, 1\}$ be the Galois Field of order 2, whose elements are residue classes modulo 2. The arithmetic operations of addition and multiplication in GF(2) are defined modulo 2. The set of all polynomials in one variable t with coefficients in GF(2),

called polynomials over GF(2), is a ring considering the arithmetic operations of addition and multiplication modulo 2. If $f(t) = a_n t^n + a_{n-1} t^{n-1} + \ldots + a_1 t + a_0$ is a polynomial over GF(2), where $a_n \neq 0$, n is defined as the degree of $f(t)$, denoted by $deg(f)$. The length of $f(t)$, denoted by $len(f)$, is defined by $len(f) = deg(f) + 1$.

Euclidean Division Theorem for Polynomials: Let $f(t)$ and $g(t)$ be polynomials over GF(2) , where $g(t) \neq 0$. There exist unique polynomials $q(t)$ and $r(t)$ over GF(2) such that $f(t) = g(t).q(t) + r(t)$, where either $r(t) = 0$ or $deg(r) < deg(g)$. The polynomial $r(t)$ is called the remainder of the division of $f(t)$ by $g(t)$, and will be denoted by $< f(t) >_{g(t)}$.

Polynomial congruence: Given $f(t)$, $g(t)$, and $h(t)$ polynomials over GF(2), we say that $f(t)$ is congruent to $h(t)$ modulo $g(t)$, and write $f(t) \equiv h(t) \ mod \ g(t)$, if $h(t) =< f(t) >_{g(t)}$.

Irreducible Polynomials: A non-zero polynomial $g(t)$, is called a divisor of $f(t)$ over GF(2) if $f(t) = a(t).g(t)$, for some polynomial $a(t)$ over GF(2). Two polynomials $f(t)$ and $g(t)$ over GF(2) are coprime if their only common divisor is 1. A non-constant polynomial $f(t)$ over GF(2) is called irreducible over GF(2) if its only divisors are possibly a constant polynomial and itself.

Chinese Remainder Theorem (CRT) for polynomials: Let $s_1(t), s_2(t), \ldots, s_N(t)$ be monic pairwise coprime polynomials over GF(2) and let $M(t) = \prod_{i=1}^{N} s_i(t)$. There exists a unique polynomial $R(t)$ over GF(2) with $deg(R) < deg(M)$, satisfying $R(t) \equiv o_i(t) \ mod \ s_i(t)$, for $i = 1, 2, ..., N$, where:

$$R(t) = < \tilde{R}(t) >_{M(t)} \quad (1)$$

$$\tilde{R}(t) = \sum_{i=1}^{N} o_i(t) \cdot m_i(t) \cdot n_i(t) \quad (2)$$

$$m_i(t) = M(t)/s_i(t) \quad (3)$$

$$n_i(t) \cdot m_i(t) \equiv 1 \quad mod \quad s_i(t) \quad (4)$$

The computation of $n_i(t)$ can be implemented using the Extended Euclidean Algorithm, which basically consists in applying the Euclidean Division Theorem several times. The algorithm complexity for computing $R(t)$ is $\mathcal{O}\big(len(M)^2\big)$ [9].

IV. PolKA Proposal

This section proposes PolKA, a polynomial RNS-based SR scheme for network fabrics that explores the polynomial CRT. It also provides an usage example and a scalability analysis.

A. Source routing scheme

PolKA architecture is composed of three elements: (i) edge nodes that embed the route labels into the packets, (ii) core nodes that execute basic packet transport, and (iii) a logically centralized SDN Controller that selects routing paths and configures the nodes. In this architecture, the SR relies on three polynomial identifiers over GF(2): (i) *routeID*: a route identifier, calculated by the controller using the polynomial CRT and embedded into the packet by the edge nodes; (ii) *nodeID*: an identifier previously assigned to core nodes by the controller in a network configuration phase; and (iii) *portID*: an identifier assigned to the output ports of each core node.

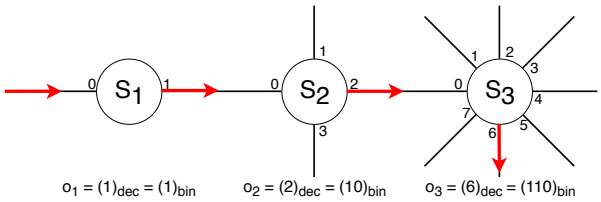


Figure 2: Example of PolKA SR.

In this paper, all polynomials will be considered as polynomials over GF(2). Note that a polynomial $f(t) = a_n t^n + a_{n-1} t^{n-1} + \ldots + a_1 t^1 + a_0 t^0$ can be represented by the bit string $a_n a_{n-1} \ldots a_1 a_0$. Thus, an identifier is represented by a bit string formed by the coefficients of a polynomial, which are either 0 or 1. Also, the bit length of the identifier is $len(f)$.

Consider a packet should be routed via a selected path, represented by N core nodes and their respective output ports.

Let S = $\{s_1(t), s_2(t), \ldots, s_N(t)\}$ be the multiset of the polynomials representing the *nodeID*s of the nodes in this path. The set S must be composed of pairwise co-prime polynomials, and satisfy the condition $deg(s_i(t)) \geq \lceil \log_2(nports) \rceil$, where $nports$ denotes the number of ports in the node. In this paper, we assume that $s_i(t)$ are irreducible polynomials.

Let O = $\{o_1(t), o_2(t), \ldots, o_N(t)\}$ be the multiset of N polynomials, where $o_i(t)$ represents the output port for the packet at the core node $s_i(t)$, for $i = 1, 2, \ldots N$, satisfying the condition that $deg(s_i) > deg(o_i)$. For instance, if the output port polynomial is $o_i(t) = 1 \cdot t^2 + 1 \cdot t$, it maps the port 110 and the packet is forwarded to port label 6 at node $s_i(t)$.

Based on the definition of the path represented by S and O, the Controller calculates the *routeID* using the polynomial CRT (see Section III) as the polynomial $R(t)$ that satisfies:

$$R(t) \equiv o_i(t) \mod s_i(t), \quad for \quad i = 1, 2, \ldots N \quad (5)$$

The *routeID* is embedded in the packet by the edge element, and the forwarding operation in each core node calculates the output port as the remainder of the euclidean division of the *routeID* in the packet by its *nodeID*: $o_i(t) = < R(t) >_{s_i(t)}$

The Controller may proactively compute $R(t)$ or calculate it when the first packet of a flow arrives. On the other hand, core nodes only execute a simple mod operation per packet.

B. Usage example

Fig. 2 shows a usage example in a topology composed of three core nodes. The degrees of the polynomials of *nodeID*s s_1, s_2, and s_3 must be equal or greater than 1, 2, and 3, in order to encode 2, 2^2, and 2^3 ports, respectively. Let the following irreducible polynomials be assigned to s_i:

$$s_1(t) = t + 1 = 11$$
$$s_2(t) = t^2 + t + 1 = 111$$
$$s_3(t) = t^3 + t + 1 = 1011$$

Considering the path defined in Fig. 2 ($s_1 \rightarrow s_2 \rightarrow s_3$), the output port set O is composed by the polynomials:

$$o_1(t) = 1, \quad o_2(t) = t = 10, \quad o_3(t) = t^2 + t = 110$$

Algorithm 1 Computation of the maximum $len(R)$.

```
1: function MAXLEN(nports, diameter, size)
2:     mindeg ← ⌈log_2(nports)⌉
3:     nodelist ← generate_irred_poly_list(mindeg, size)
4:     pathlist ← get_last_itens(nodelist, diameter)
5:     length ← 0
6:     for elem ∈ pathlist do
7:         length ← length + deg(elem);
8:     return length                          ▷ Maximum len(R)
```

Thus, $R(t)$ must satisfy the conditions of Eq. (5):

$$R(t) \equiv 1 \mod (t+1)$$
$$R(t) \equiv t \mod (t^2+t+1)$$
$$R(t) \equiv (t^2+t) \mod (t^3+t+1)$$

As in CRT for polynomials (Section III), we have:

$$M(t) = (t+1)\cdot(t^2+t+1)\cdot(t^3+t+1)$$
$$m_1(t) = s_2(t)\cdot s_3(t) = (t^2+t+1)\cdot(t^3+t+1)$$
$$m_2(t) = s_1(t)\cdot s_3(t) = (t+1)\cdot(t^3+t+1)$$
$$m_3(t) = s_1(t)\cdot s_2(t) = (t+1)\cdot(t^2+t+1)$$

And solving Eq. (4), we find the polynomials $n_i(t)$:

$$n_1(t) = 1, \quad n_2(t) = 1, \quad n_3(t) = t^2+1$$

Finally, we can calculate $R(t)$ according to Eq. (1):

$$\tilde{R}(t) = (t^2+t+1)(t^3+t+1) + t(t+1)(t^3+t+1) + (t^2+t)(t+1)(t^2+t+1)(t^2+1) = t^7+t^6+t^5+t^2+1$$

$$R(t) = <\tilde{R}(t)>_{M(t)} = t^4 = 10000$$

Therefore, packets should embed the *routeID* 10000, so each node can calculate its *portID* by dividing this *routeID* by its own *nodeID*. For example, the remainder of $R(t) = 10000$ divided by $s_3(t) = 1011$ is $o_3(t) = 110$ (port label 6).

C. Scalability analysis of the routeID

The goal of this section is to investigate the overhead of our new scheme in the bit length of *routeID* in comparison to the Port Switching and integer RNS-based approaches. The bit length of $R(t)$, $len(R)$, in PolKA is given by the equation:

$$len(R) = len(<\tilde{R}(t)>_{M(t)}) \leq \sum_{i=1}^{N} deg(s_i) \qquad (6)$$

Algorithm 1 shows a pseudo-code for computing the maximum $len(R)$, given: the number of ports in each node ($nports$), the number of nodes ($size$), and the topology diameter ($diameter$). For the sake of simplicity, we consider all nodes have the same number of ports. A list of *nodeID* polynomials ($nodelist$) is generated, which consists of $size$ irreducible polynomials with degree greater than or equal to the minimum degree ($mindeg$). Note that we select polynomials with the lowest possible degree (e.g., if $mindeg = 5$, we start assigning one of the 6 existing irreducible polynomials of degree 5 to nodes, and, if necessary, we use the 9 existing irreducible polynomials of degree 6, and so forth). Thus, $nodelist$ is already ordered by degree. Finally, we select the very worst case scenario in which the polynomials in $nodelist$ with the greatest degrees assigned to the nodes in the longest possible path (i.e., the diameter). To this end, we pick the x last elements of $nodelist$, where $x = diameter$, and calculate the maximum $len(R)$ according to Eq.(6).

Table II: Maximum $len(R)$ for example topologies.

Topology	$nports$	$diam.$	$size$	Bits for PolKA	Bits for Port Swit.
Two-tier S16 L16*	24	3	32	21	15
Fat-tree 16 pods	16	5	320	55	20
ARPANET	4	7	20	42	14
GEANT2	8	7	30	49	21

** Two-tier topology with 16 spine switches and 16 leaf switches.*

Table II compares the scalability of PolKA and Port Switching for two common DC topologies (fat-tree and two-tier), and two continental backbone topologies [21] (ARPANET and GEANT2). This topology set covers diverse properties for number of ports, diameter and size. For backbone topologies, as the number of ports varies per node, we considered $nports$ as the maximum number of ports of any node.

The results for the integer RNS-based approach were omitted, because they are very close to PolKA. This means that using PolKA instead of the integer version does not incur in reserving extra bits for the *routeID* header. The Port Switching method consumes less bits for representing the *routeID* than RNS-based SR approaches, specially in the cases where the size of the topology is high (e.g., fat-tree). These cases demand a large number of irreducible polynomials for PolKA, causing the selection of polynomials of high degree or large integer numbers even when the node does not demand many ports.

For the topologies under worst case analysis in Table 1, the maximum $len(R)$ results show that PolKA fits existing packet headers (e.g., 96 bits of Ethernet source and destination addresses, or a stack of MPLS labels with 20 bits per label). When deploying RNS-based SR, the cost to exploit RNS features may be to reserve more bits in the packet header for representing the *routeID*. Nevertheless, there are known techniques that make optimal assignment of *nodeID*s (avoiding the worst case scenario), and reduce *routeID* length [12], [14].

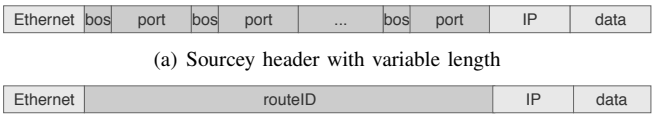
V. Implementation Design

This section studies how to implement PolKA and Port Switching according to P4 architecture [11]. From now on, we will refer to Port Switching as Sourcey [5], as it is a prominent example of the method. Design choices may change according to application requirements and target platform features. For example, the *routeID* can be added in a new or existing header, while the size of headers depend on the network topology.

A. Sourcey pipeline

Our implementation of Sourcey creates a new header that includes the port stack after the Ethernet header, as shown in Fig. 3(a). Each item of the stack has a `bos` (bottom of stack) bit and a port number. The `bos` bit is 1 only for the last entry.

When the packet reaches an edge switch from an end-host, the `etherType` in the Ethernet header is TYPE_IPv4=`0x800`. Thus, the switch must parse the



(a) Sourcey header with variable length

(b) PolKA header with fixed length

Figure 3: Sourcey and PolKA headers.

IPv4 header, encapsulate the SR header, and change the `etherType` to TYPE_SR=`0x1234`.

Each edge switch has a table, which is populated by the controller and maps destination IP address to their routing paths, represented by a port list. The result of this table lookup is an action that sets the output port to the directly connected core switch and encapsulates n SR headers, where n is the number of core switches in the path. At core switches, the pipeline of Fig. 4(a) is executed. Firstly, the Ethernet header is parsed to check if the `etherType` is TYPE_SR. Then, the switch parses the first SR header, gets the output port, and pops this header from the stack. If the `bos` bit is one, it is the last hop and the `etherType` is changed to TYPE_IPv4. Another possible implementation for progressing in the list is to increment an index of the current position in each hop.

B. PolKA pipeline

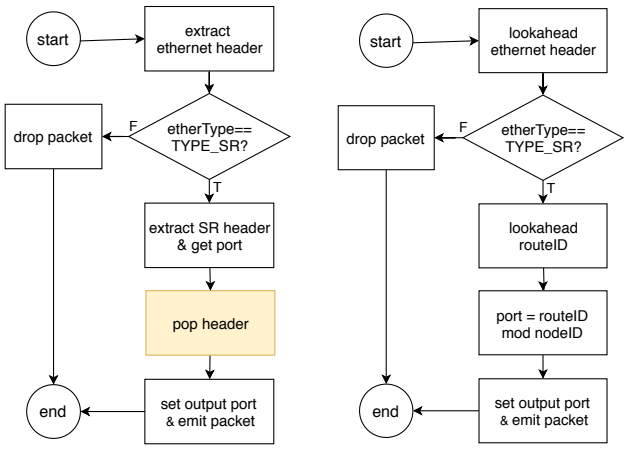
The PolKA header contains a *routeID*, whose maximum length depends on the network topology, as explained in Section IV-C. Fig. 3(b) shows the format of PolKA header with a fixed length field for storing the *routeID*, after the Ethernet header. At edge switches, PolKA pipeline for encapsulating the SR header is similar to Sourcey, but the result of the table lookup is an action that sets the output port to the directly connected core switch and encapsulates a single *routeID*.

At core switches, the pipeline of Fig. 4(b) is executed. When `etherType` is TYPE_SR, as PolKA only needs read access to packet headers, it uses the `lookahead` method of P4 language that evaluates a set of bits from the input packet without advancing the packet index pointer. To discover the output port, the switch has to perform a $\mod$ operation between the *routeID* in the packet and its own *nodeID*.

Besides, in PolKA, there is no information in the header to identify the last hop. This is not a problem in fabric networks, because the packet delivery to an edge switch represents the end of the SR path. At the edge switch, the SR header is removed and the etherType is changed to TYPE_IPv4.

C. Comparison between Sourcey and PolKA pipelines.

- Sourcey header has variable size, depending on the number of remaining hops in the path. On the other hand, PolKA has a fixed length header to store the *routeID*;
- In P4, Sourcey needs to create one encapsulating action for each stack size (e.g., `add_header_1hop`, `add_header_2hops`, ...), which increases the number of code lines and memory for deploying the edge code;
- In Sourcey, each core switch performs a header rewrite to update the stack when performing the pop operation (yellow in Fig. 4(a)). On the other hand, in PolKA (Fig. 4(b)), the packet remains unchanged along the path;
- In Sourcey, the output port is directly available in the SR header, while PolKA requires an arithmetic operation over the *routeID* to calculate the output port.



(a) Sourcey (b) PolKA

Figure 4: P4 pipelines for core switches.

Thus, if we can perform the $\mod$ operation of PolKA with equivalent latency to the rewrite operation of Sourcey, we could take advantage of RNS properties without compromising performance, when compared to Port Switching.

D. Reuse of CRC for implementing mod operation in P4

Polynomial or integer $\mod$ operations with non-constant operands are not natively supported by commodity network hardware and are not available in P4 language. Thus, as stated before, one of the main contributions of this work is to develop a technique that allows the execution of the polynomial $\mod$ in hardware by reusing common CRC operations.

In CRC operations, sender and receiver agree on a generator polynomial (G), which is a $r+1$ bit pattern used for error-detection. For data D with d bits, the sender calculates additional r bits (R), and appends them to D in such a way that the result is a polynomial with $d+r$ bits that is divisible by G using modulo-2 arithmetic [16]. Thus, the CRC code is the remainder of D shifted left by r bits, divided by G: $R =< D \cdot 2^r >_G$. Therefore, we could try to map the *routeID* as D, and the *nodeID* as G. However, PolKA does not perform a shift operation over the *routeID* as done in the CRC strategy.

As the degree of G is r, this problem can be solved if we separate the *routeID* in two parts: *routeID* $= D * 2^r + dif$, where dif is the r least significant bits of the *routeID*. Firstly, we shift right the *routeID* by r bits to produce the data D, which will be the input of the CRC function. Then, the bits that were lost with the shift right operation (dif) can be added back to the calculated CRC remainder in the end of the computation to produce the output port. Since the degree of the *portID* obtained is less than r, the unicity property in division algorithm for polynomials assures that the outcome polynomial coincides with the remainder obtained by direct

division of *routeID* by the *nodeID*. Also, in binary arithmetic, both the addition and subtraction operations are identical to the logical XOR operation. These steps are described as follows:

1) $G = nodeID$, $r = degree(G)$
2) $D = routeID \div 2^r$ ***(SHIFT RIGHT)***
3) $dif = routeID - D * 2^r$ ***(SHIFT LEFT, XOR)***
4) $R =< D * 2^r >_G$ ***(CRC)***
5) $portID = dif + R$ ***(XOR)***

Therefore, the switch calculates the output port by using two *SHIFT*, one CRC, and two *XOR* operations, which is more computationally efficient than executing a division.

With the use of this technique, PolKA can be deployed in P4 targets that allow the configuration of generator polynomials. Since P4 supports CRC operations through the use of external libraries [11], the support for customized polynomials depends on the architecture models and how specific targets implement them. The `PSA`[2] and `v1model`[3] architectures support customized polynomials of 16 and 32 bits. In terms of targets, the software switch `bmv2`[4] and the hardware switch Tofino from Barefoot support customized CRC polynomials, while Netronome SmartNICs only support fixed CRC polynomials.

VI. Proof-of-concept Prototypes

Two proof-of-concept prototypes were implemented to evaluate the main functionalities of PolKA in comparison to Sourcey: (i) an emulated setup to evaluate end-to-end scenarios, and (ii) a physical setup that uses Netronome SmartNICs[5] to evaluate forwarding in a single hop scenario.

A. Emulated prototype

1) P4 architecture and target: The software switch `bmv2 simple_switch` with the `v1model` architecture was selected as the target for this prototype, because it supports all the functionalities required by PolKA, such as the configuration of CRC polynomials. It is important to highlight that this software switch is a user space implementation with focus on feature testing. There are other high performance implementations of P4 software switches and compilers (e.g., PISCES[6], P4ELTE[7], and MACSAD[8]), but they do not yet cover all the features required by PolKA. As these implementations evolve, it will be possible to test our prototype with higher loads. For the time being, the solution was to limit the link rates to 10Mbps in our emulated prototype to avoid reaching the processing capacity limits of `bmv2 simple_switch`.

2) Setup description: The setup consists of one server Dell PowerEdge T430, with one Intel Xeon E5-2620 v3 2.40GHz processor, and 64GB of RAM. To build our emulated environment, we used a branch of `p4app`[9], which is a tool that creates a container with an emulated network using `Mininet`

[2]https://p4.org/p4-spec/docs/PSA.html
[3]https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4
[4]https://github.com/p4lang/behavioral-model
[5]https://www.netronome.com/
[6]http://pisces.cs.princeton.edu/
[7]http://p4.elte.hu/
[8]https://github.com/intrig-unicamp/macsad/
[9]https://github.com/p4lang/p4app/tree/rc-2.0.0/

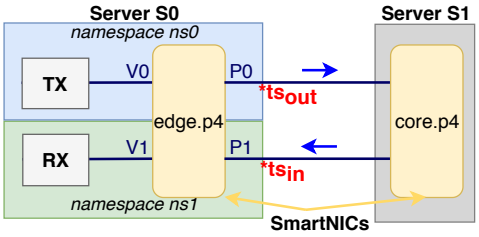


Figure 5: SmartNIC setup.

and allows to run different P4 programs on different switches. This functionality is crucial to emulate fabric networks, which require different P4 programs for edge and core elements.

3) Control plane implementation: It has two main functionalities: (i) for core: compute *nodeID*s, and configure core switches with their respective identifiers; and (ii) for edge: compute routing paths for the traffic flows, calculate *routeID*s for these paths, and configure table entries in edge switches that will be responsible for encapsulating *routeID*s. For the RNS computation of *nodeID*s and *routeID*s, as described in Section IV-A, we developed a Python application that uses the package `galoistools` of the `sympy` library[10] for GF(2) arithmetic operations. For programming each core and edge switch, we developed a control plane application in Python that communicates with the switches using the CLI commands provided by the `bmv2 simple_switch`. The format of the control messages is defined by an API that connects to a Thrift RPC server running in each switch process.

4) Header size: The `bmv2 simple_switch` only supports the specification of CRC polynomials of 16 and 32 bits. Therefore, although our tests could use much smaller degrees, our PoC had to adopt polynomials of degree 16. As the diameters of our test topologies are smaller than 10, the size of the PolKA header was defined as 160 bits. To get a fair comparison, we defined the Sourcey header as 16 bits (bos and port), so, for 10 hops, the array of headers has 160 bits.

B. Hardware prototype with SmartNICs

In the emulated prototype, the CRC operation is executed in software using CRC tables. However, the main benefit of using CRC is the execution of the `mod` operation in hardware with better performance. To this end, we built a hardware prototype.

1) Setup description: The setup is illustrated by Fig. 5 and consists of two servers: (i) S0: a device under test (DUT), running the core functionalities; and (ii) S1: a traffic generator (TG), running the edge functionalities, and transmitter (TX) and receiver (RX) functionalities in separate network namespaces. Both servers are Dell PowerEdge T430, with one Intel Xeon E5-2620 v3 2.40GHz processor, 16GB of RAM, and one Netronome Agilio CX 2x10GbE SmartNIC.

2) SmartNICs features: Netronome SmartNICs give access to hardware timestamps to P4 programs.They partially implement the functionalities of *v1model* for P4 16, but it currently supports only a small set of fixed CRC polynomials, which restricts the use of its CRC hardware in PolKA. Nevertheless,

[10]https://www.sympy.org/

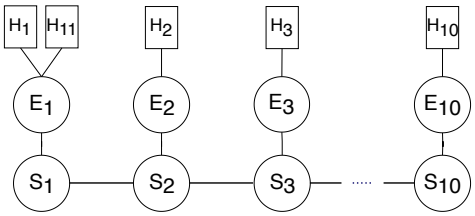


Figure 6: Linear fabric topology.

we could use them for measuring forwarding latency in one core node, as proposed in the next section.

3) P4 programs: The P4 programs are adaptations of the codes used in the emulated prototype. The new edge code encapsulates both SR headers and a timestamp header for executing latency measurements, which is composed of an egress (ts_{out}) and an ingress timestamp (ts_{in}).

At server S0, packets are generated at TX and forwarded to the SmartNIC, where the edge adds SR (PolKA or Sourcey) and timestamp headers. When the packet leaves the edge, the hardware timestamp is assigned to ts_{out}. Then, it is transmitted to server S1 and processed by the core code at the SmartNIC, which is responsible for parsing the SR header and computing the output port. Since it is not possible to configure customized CRC polynomials in the SmartNICs, we use a standard CRC operation with a fixed CRC-16 polynomial. In this way, we execute all the PolKA pipeline steps (including the CRC operation) to measure their contribution in the total latency, and select a fixed output port. As a result, the packet is delivered to server S0, where the edge code assigns the value of the hardware timestamp to ts_{in}, removes the SR header, and delivers the packet to RX. Finally, all packets are captured with `tcpdump` tool, and parsed offline to extract the core forwarding latency for each packet as: $ts_{in} - ts_{out}$.

VII. Evaluation

This section evaluates PolKA and Sourcey for: (i) end-to-end tests in the emulated prototype, and (ii) single hop latency test in the hardware prototype. Also, it demonstrates path migration and failure reaction features in PolKA. For the tests, we considered Ethernet frames of 98 Bytes as small packets, and frames of 1242 Bytes as big packets. The average and standard deviation are presented in all the results.

A. Emulated prototype: End-to-end tests in PolKA vs. Sourcey

The test uses the linear fabric topology of Fig. 6 to compare PolKA and Sourcey as the number of hops increases in the core network (e.g., from 0 for path $H_1 \rightarrow H_{11}$ to 9 for path $H_1 \rightarrow H_{10}$). The following experiments were executed: (i) round trip time (RTT): host H_1 sends 1 ICMP packet/s during 60s to each of the other hosts using `ping` tool; (ii) jitter: host H_1 sends an UDP traffic of 5Mbps (half of link capacity) with big packets to each of the other hosts during 60s using the `iperf` tool; and (iii) flow completion time (FCT): host H_1 transmits a file of 100Mb with big packets over a TCP connection to each of the other hosts using the `iperf` tool (3 repetitions). Fig. 7 shows the comparison between PolKA and Sourcey solutions for RTT, jitter, and FCT experiments.

In the RTT experiments (Fig. 7(a) and Fig. 7(b)), it is possible to observe that: the RTT grows linearly with the increase of the number of hops for both solutions; (ii) Sourcey solution presents better RTT performance than PolKA solution; and (iii) the standard deviation is small and in the same order of magnitude for both solutions. Besides, there is no significant difference in the results for different packet sizes in the RTT experiments. This is because `Mininet` does not consider transmission time in the emulation. In addition, jitter (Fig. 7(c)) is small and equivalent for both solutions. Finally, the FCT experiment (Fig. 7(d)) shows that both solutions require approximately the same time to transfer the file and the standard deviation is small.

The fact that Sourcey has a better RTT performance than PolKA is related to two facts: Sourcey looses one SR header per hop, so the average packet header size is smaller than the fixed header used by PolKA; and the CRC operation for PolKA in this emulated prototype is executed in software. Nevertheless, the difference between the two solutions is small and can decrease if the CRC operation is performed in hardware, as we show in the next subsection.

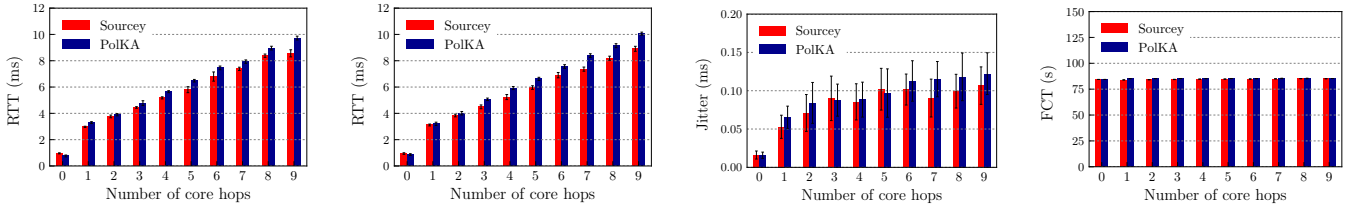
B. Hardware prototype: Core latency in PolKA vs. Sourcey

The goal is to measure the core forwarding latency for a single hop in PolKA and Sourcey when the path length increases. We consider the path length as the number of core nodes that must be included in the SR header to reach the destination. For each test execution, the traffic generator tool at TX varies the IP destination address. The last digit of the IP destination address represents the number of core nodes (e.g., if IP destination is 10.0.100.1, the number of hops to the destination is 1, while IP destination 10.0.100.9 represents 9 hops to the destination). Depending on the number of hops, the edge encapsulates the appropriate SR headers (e.g., for 5 hops, 5 SR headers in Sourcey). The following experiments were executed for small and big packets: (i) low throughput: one ICMP pps, 100 packets in total, generated with `ping` tool; and (ii) high throughput: 1Gbps UDP packets, 1000 packets in total, generated with `pktgen` tool.

Fig. 8 compares the test cases for Sourcey and PolKA. Within each test, the average latency and standard deviation in PolKA varies little when the path length increases, while in Sourcey the average latency grows linear when the path length increases. This linear increase in latency measurements for Sourcey is emphasized in the test case with high pps values (Fig. 8(c)), when the standard deviation is high for both PolKA and Sourcey due to the stress in edge and core elements. More investigation needs to be carried out in a hardware prototype that allows multi-hop tests, but the results collected so far indicate that PolKA implementation using CRC hardware is promising and can offer at least equivalent RTT and jitter performance to Sourcey.

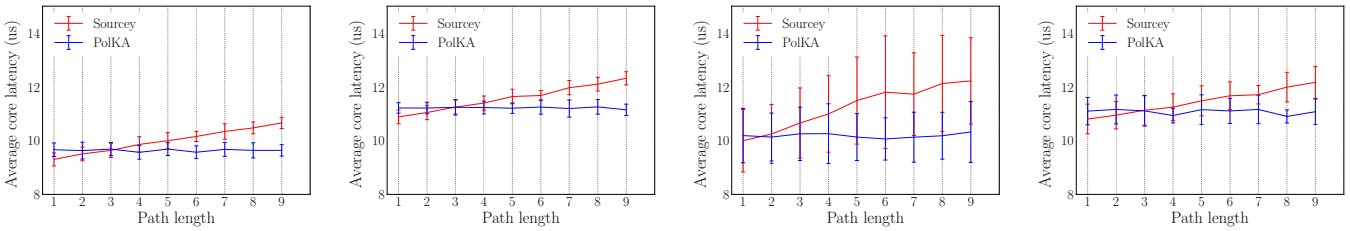
C. Emulated prototype: Agile path migration in PolKA

This experiment shows how traffic engineering can benefit from SR for bandwidth allocation with agile path migration.



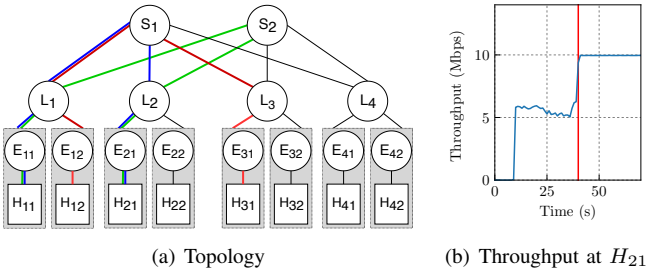
(a) RTT small packet (b) RTT big packet (c) Jitter (d) FCT

Figure 7: Linear fabric scenario: comparison between Sourcey and PolKA.



(a) Low throughput, small packets (b) Low throughput, big packets (c) High throughput, small packets (d) High throughput, big packets

Figure 8: Comparison of Sourcey and PolKA test cases.



(a) Topology (b) Throughput at H_{21}

Figure 9: Agile path migration in two-tier topology.

Fig. 9(a) shows a two-tier data center, which contains 2 spine switches (S_n) and 4 leaf switches (L_i), all running core functionalities. Each leaf switch is connected to one server, which has an edge switch (E_{ij}) to interconnect VMs (H_{ij}).

At 10s, flow A ($H_{11} \rightarrow H_{21}$) and flow B ($H_{12} \rightarrow H_{31}$) start TCP traffics with big packets using the `iperf` tool. Initially, flow A is allocated to blue path ($E_{11} - L_1 - S_1 - L_2 - E_{21}$) and flow B is allocated to red path ($E_{12} - L_1 - S_1 - L_3 - E_{31}$). Thus, these flows compete for bandwidth at link $L_1 - S_1$ in the interval of 10s to 40s. The effects of TCP congestion control for fair share of total bandwidth can be seen in Fig. 9(b) that shows the throughput at the destination of flow A (H_{21}). At 40s, the traffic engineering decides to exploit idle links and migrates flow A from the blue path to the green path ($E_{11} - L_1 - S_2 - L_2 - E_{21}$). From this moment, there is no competition with flow B, so flow A consumes all the link bandwidth.

To perform path migration, the SDN Controller only has to modify a single flow entry at the edge switch E_{11} for destination H_{21}. The only field that has to be modified is the *routeID* to embed the new route through green path. Once this single operation is executed, all the packets of flow A that leave H_{11} will be tagged with the *routeID* of the new path.

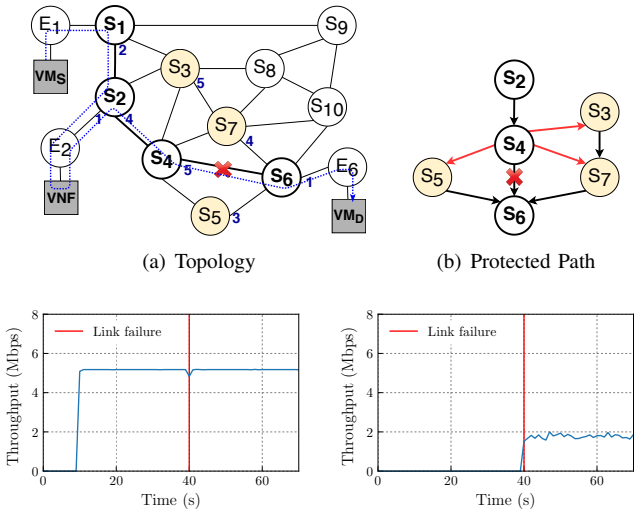
D. Emulated prototype: Use of RNS properties in PolKA

The goal of this experiment is to show an example of how PolKA can take advantage of special RNS properties. More specifically, it explores a property that states that the order of the nodes in the path is irrelevant. Based on this property, we integrate a fast failure reaction mechanism proposed by KAR [8] to the SFC scheme proposed in KeySFC [15]. KAR proposes the concept of resilient forwarding path, called protection path. The main idea is to proactively add redundant nodes in the *routeID* that are not part of the original route. When there is a link failure, packets are deviated from faulty links with routing deflections and may occasionally reach these redundant nodes, which are responsible to guide the packets back to the original route. In this way, there is no need to communicate with a controller (or the source) to compute an alternative path, because, as soon as the core node detects a failure, it randomly deflects packets to one of its healthy links.

Fig. 10(a) shows an example scenario for SFC $\text{VM}_\text{S} \rightarrow \text{VNF} \rightarrow \text{VM}_\text{D}$. The path in the core switches for the first SFC segment ($\text{VM}_\text{S} \rightarrow \text{VNF}$) is: nodes S = $\{S_1, S_2\}$ and ports O = $\{2, 1\}$. The path for the second SFC segment ($\text{VNF} \rightarrow \text{VM}_\text{D}$) is: nodes S = $\{S_2, S_4, S_6\}$ and ports O = $\{4, 5, 1\}$. These paths are called unprotected paths, because they do not add any redundant node for failure protection. Therefore, if any link of these paths fails, the packets will be dropped.

Applying the protection mechanism to generate the *routeID* of the second SFC segment ($\text{VNF} \rightarrow \text{VM}_\text{D}$), we add the extra nodes S_3, S_5, and S_7 with ports 5, 3, and 4, respectively. Thus, when link S_4-S_6 fails, S_4 deflects packets to any of its other links, and packets will be driven back to S_6, as shown in Fig. 10(b). Thus, the protected path is represented by nodes S = $\{S_2, S_3, S_4, S_5, S_6, S_7\}$ and ports O = $\{4, 5, 5, 3, 1, 4\}$. As the protected path already contains redundant nodes, no change in the *routeID* is needed when the failure happens.

To integrate KAR mechanism in our prototype, we de-



(a) Topology (b) Protected Path

(c) Throughput at destination (d) Throughput at S_3

Figure 10: Fast failure reaction for failure of link S_4-S_6.

veloped a simple control plane application that causes link failures and makes port failure information available to the data plane by populating a table of faulty ports. In addition, we modified the core pipeline to perform a lookup in this table before sending the packet to the output port. If there is a hit, the packet is randomly deflected to one of the other healthy ports. Otherwise, the packet is transmitted normally. The generation of a random value within an interval is provided by `v1model` and could also be replaced by a hash function if the objective is to always select the same port per flow. Failure detection mechanisms are not in the scope of this work.

Fig. 10 shows throughput measurements using the `bwm-ng` tool at VM_D (Fig. 10(c)) and S_3 (Fig. 10(d)). Results for S_5 and S_7 were omitted, because they are similar to Fig. 10(d) as the traffic was uniformly deflected through S_3, S_5, and S_7 after the failure. At 10s, we start a 5Mbps UDP traffic from VM_S to VM_D. At 40s, the link S_4-S_6 is disconnected. At VM_D, the traffic perceives a small loss until the failure is signalized by the control plane and deflections start. Therefore, our scheme was able to react to failures without any packet modification as the redundant nodes were already included in the *routeID*.

VIII. Conclusion

Herein, a binary polynomial representation of a fully stateless RNS-based SR mechanism, called PolKA, was proposed, implemented, and evaluated. To the best of our knowledge, this is the first work to apply the CRT theorem in conjunction with finite fields polynomials to solve routing problems and it is one of the main contributions of this paper. Moreover, our P4-based emulated and hardware prototypes demonstrated that is feasible to deploy RNS-based SR in commodity network equipment by reusing CRC hardware, with performance equivalent to traditional Port Switching approaches. This achievement has the potential to enable a new range of complex network applications that explore RNS intrinsic features, such as fast failure reaction and route authenticity.

Future works include extensions of our hardware prototype for multi-hop scenarios using the Tofino switch, and implementation of a multicast solution. Moreover, the polynomial arithmetic of PolKA in hardware description languages has the potential to synthesize RNS-based SR in smaller chip areas and reduced clock cycles. Finally, we plan to extend our polynomial scheme using GFs of higher orders.

Acknowledgment

This study was financed by CAPES (Finance Code 001), CNPq, FAPES, CTIC, RNP, and EU H2020 (FUTEBOL).

References

[1] M. Casado *et al.*, "Fabric: A retrospective on evolving sdn," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 85–90.

[2] M. Alizadeh *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, p. 503–514, Aug. 2014.

[3] S. Hu *et al.*, "Explicit path control in commodity data centers: Design and applications," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. USA: USENIX Association, 2015, p. 15–28.

[4] S. A. Jyothi *et al.*, "Towards a flexible data center fabric with source routing," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research*. New York, NY, USA: ACM, 2015, pp. 10:1–10:8.

[5] X. Jin *et al.*, "Your data center switch is trying too hard," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research*. New York, NY, USA: ACM, 2016, pp. 12:1–12:6.

[6] M. Soliman *et al.*, "Source routed forwarding with software defined control, considerations and implications," in *ACM Conference on CoNEXT Student Workshop*. New York, NY, USA: ACM, 2012, pp. 43–44.

[7] H. Wessing *et al.*, "Novel scheme for packet forwarding without header modifications in optical networks," *Journal of Lightwave Technology*, vol. 20, no. 8, pp. 1277–1283, Aug 2002.

[8] R. R. Gomes *et al.*, "KAR: Key-for-any-route, a resilient routing system," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, June 2016, pp. 120–127.

[9] V. Shoup, *A computational introduction to number theory and algebra*. Cambridge university press, 2009.

[10] M. Martinello *et al.*, "KeyFlow: a prototype for evolving SDN toward core network fabrics," *IEEE Network*, vol. 28, no. 2, pp. 12–19, 2014.

[11] The P4 Language Consortium, "P4 16 Language Specification." [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[12] Y. Ren *et al.*, "Flowtable-free routing for data center networks: A software-defined approach," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, Dec 2017, pp. 1–6.

[13] W. Jia, "A scalable multicast source routing architecture for data center networks," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 1, pp. 116–123, January 2014.

[14] A. Liberato *et al.*, "RDNA: Residue-Defined Networking Architecture Enabling Ultra-Reliable Low-Latency Datacenters," *IEEE TNSM*, Nov 2018.

[15] C. K. Dominicini *et al.*, "KeySFC: Traffic steering using strict source routing for dynamic and efficient network orchestration," *Computer Networks*, vol. 167, p. 106975, 2020.

[16] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, Jan 1961.

[17] M. Grymel and S. B. Furber, "A novel programmable parallel crc circuit," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 10, pp. 1898–1902, Oct 2011.

[18] Microchip, "32-bit programmable cyclic redundancy check (crc)," http://ww1.microchip.com/downloads/en/DeviceDoc/dsPIC33_PIC24-FRM,-32-Bit-Programmable-Cyclic-Redundancy-Check-(CRC)-DS30009729C.pdf, 2018, Accessed: 2019-01-13.

[19] J. C. Bajard, "A residue approach of the finite fields arithmetics," in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, Nov 2007, pp. 358–362.

[20] I. Herstein, *Topics in Algebra*, 2nd ed. John Wiley & Sons, 1975.

[21] S. K. Routray *et al.*, "Statistical model for link lengths in optical transport networks," *J. Opt. Commun. Netw.*, vol. 5, no. 7, pp. 762–773, Jul 2013.